Tinmith-evo5
A Software Architecture for Supporting Research Into Outdoor Augmented Reality Environments

Wayne Piekarski and Bruce H. Thomas
Wearable Computer Laboratory
School of Computer and Information Science
University of South Australia
Mawson Lakes, SA, 5095, Australia
{wayne, thomas}@cs.unisa.edu.au

Abstract

This paper presents a new software architecture we have developed, known as Tinmith-evo5, which is designed to streamline the process of writing applications for augmented reality and other types of virtual environments. Writing AR applications can be a complex and time consuming task, as there is little existing technology to support this process. This lack of technology requires the implementation of most systems from the ground up. There are many ways of designing these systems, each having its own set of trade-offs. We present Tinmith-evo5 as an architecture to handle this complexity, and have found it to be highly effective for our investigations.

Keywords: augmented reality, virtual reality, 3D user interfaces, software architecture

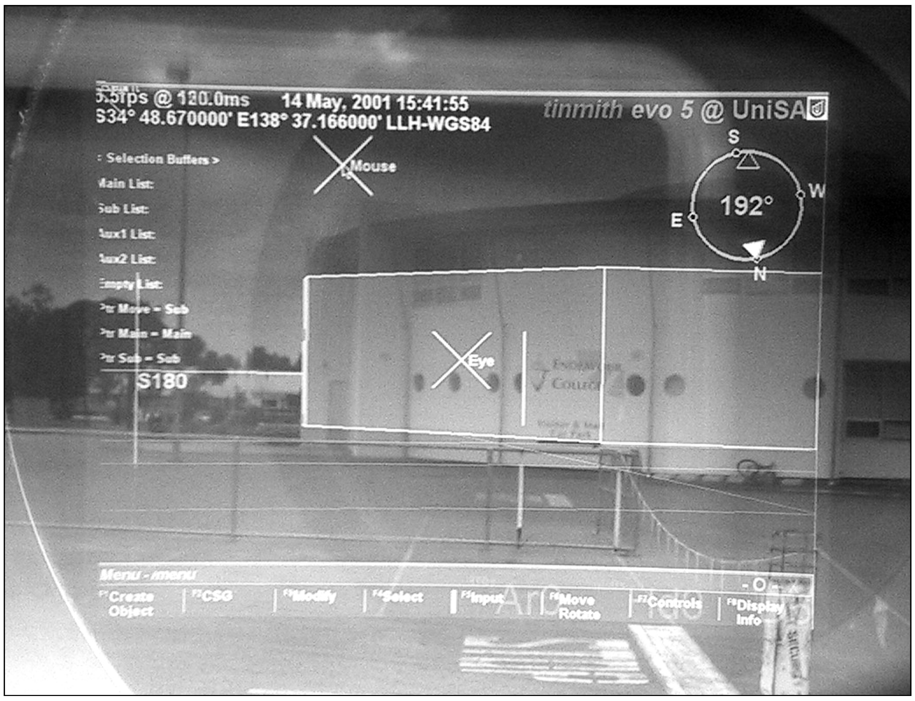

Figure 1 - Tinmith-Metro 3D building construction example, with user interface menu and situational awareness gadgets, demonstrating the Tinmith-evo5 architecture described in this paper

1 Introduction

Over the last few years, we have developed a number of augmented reality applications, such as the example Tinmith-Metro application [PIEK01b] shown in Figure 1. As part of our investigations into augmented reality and user interfaces, we have used various designs and methodologies to construct our systems in an efficient manner. These ideas have been iteratively improved over time; and we present the culmination of this process, the Tinmith-evo5 architecture. This architecture comprises a number of components, including an overall object-oriented methodology, libraries of components that can be reused for system functionality, and the software to implement complete applications. Although useable for a wide range of tasks, the main focus was to be able to easily build virtual environments. It is not designed as an architecture for wearable context awareness or other high level information sharing.

For traditional 2D desktop environments, many stable design methodologies, toolkits, libraries, and input devices are available. Since virtual environments (VEs), and augmented reality (AR) in particular, are relatively new fields, the same support is not mature enough for implementing new VE applications. This paper presents our approach to writing AR applications. We intend for our work to serve as a guide to follow for implementing applications to meet similar requirements.

The remainder of this section will describe an overview of Tinmith-evo5, and what the new and novel contributions are. We then discuss our requirements and reasoning behind the design. The next section discusses previous work in the area, comparing that work against Tinmith-evo5, and how the current design evolved from our previous implementations. The details of Tinmith-evo5 are then presented as follows: the foundation concepts, support for input devices, rendering technology, and the modelling system. Finally, we conclude the paper with a demonstration of applications we have implemented with the system.

1.1 Contribution

This paper presents our novel architecture for constructing AR applications, as well as virtual environments in general. As with the design of any system, there are trade-offs that must be made. Many strive to achieve a theoretically pure solution, which is designed to handle all possible situations - we take a different approach. We believe architectures that are too generic can make implementing functional applications difficult. We narrowed down the focus of our architecture to ensure that it would be simpler to implement, and useful for the tasks we wanted to perform. The architecture is still expressive enough to provide a platform for a wide range of AR and VE applications.

Tinmith-evo5 is an architecture for the construction of applications for virtual environments; it describes many levels in the design and implementation process. Data flow is the core concept of the architecture (see Figure 2), where sensor data arrives in the system, is processed through a series of layers, and then is rendered to the display. Tinmith-evo5 defines all of these processing layers, providing a complete solution for building virtual environments.

Using the data flow methodology and the techniques we have devised, it is possible to easily build highly complex virtual environment applications. By writing various objects that perform operations on data, these are combined together to implement higher-level software entities such as 3D renderers, geometric modelling engines, user interfaces, tracker abstractions, and other components that are needed for virtual environments.

Overall, Tinmith-evo5 is a system that facilitates the implementation of 3D virtual environment applications with a similar level of support as supplied in a 2D desktop environment.

To achieve this facilitation, Tinmith-evo5 provides support for a wide range of implementation areas.

Tinmith-evo5 was designed and implemented to support our investigations into user interfaces for outdoor AR systems. As an example of using Tinmith-evo5, we have implemented the Tinmith-Metro application. Tinmith-Metro allows a user to perform 3D interactive modelling of buildings outdoors, using custom designed 3D input devices and new interaction techniques.

1.2 Application requirements

Toolkits and design methodologies are useful when they solve some particular problem domain. As previously stated, our application domain is mobile augmented reality applications, although we extend this domain to include virtual environments in general. This extension does not complicate the architecture further and extends its usefulness to other areas.

Simple AR applications are display oriented only. These systems take in head tracker information, process it, and then render new information as an image out, allowing the user to experience the VE. These applications are usually straightforward to implement, as they use conventional renderers, with the head tracker controlling the user's point of view. The first application requirement is to provide data flow from the head tracker to the display.

More complicated AR applications allow the user to interact with the system (both to control the application itself, and the data presented by the application). This involves multiple different input devices. The next set of requirements is for input device abstraction, state machines for tracking changes, and a dynamic renderer. The renderer should support a scene graph of objects, as well as the ability to perform powerful 3D modelling techniques.

1.3 Low level requirements

The two starting points for our design are an object oriented approach and a data flow model. We employ an object oriented approach to help cope with the problems of implementing large complicated software systems. By breaking our tasks down into small objects, each one can solve a small problem, and then be combined together to solve larger problems. By allowing these objects to communicate with each other, data can then flow through the system - from the tracking device input to the output generated by the renderer (see Figure 2).

Libraries of commonly used functions are needed to assist with the implementation of the objects. In many cases, existing APIs for the operating system and C library have interface problems that we would prefer to develop solutions to only once. Abstraction layers are required to make the system portable to various hardware and operating systems types. For example, tracking hardware has not been standardised, so the communications still vary widely. An abstraction for these trackers is required for orientation and position information.

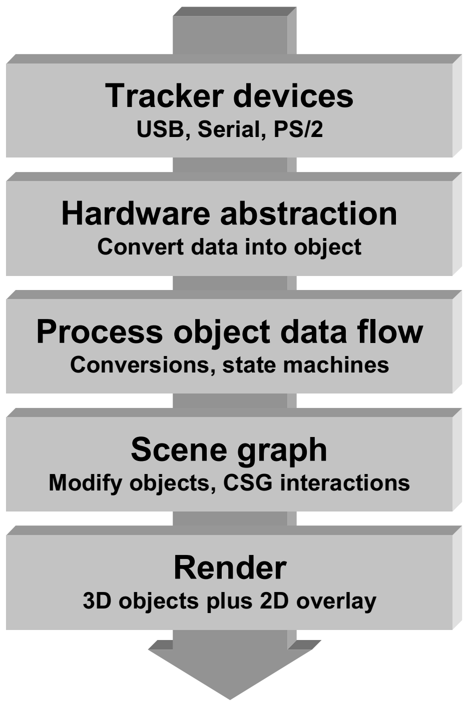

Figure 2 - System data flow

The object oriented classes of the system should be defined in a consistent fashion, to facilitate their combination in interesting and previously unthought of ways. The data flow model is designed to allow for this flexibility.

The design should allow abstraction layers to be bypassed if required. This allows for a trade-off during the implementation phase between the benefits of adhering to the design model and the construction of workable solutions. The programmer decides how to implement the solution in order to make their task simpler, as there may be special cases that need to be catered for. By allowing programmers to access object variables directly, it is also possible to support legacy code using class wrappers (this feature allowed us to leverage our work from previous systems). Fundamentally, an architecture should not force the application to pay a performance penalty for features not required at a given instance – resources in mobile computers are at a premium, and so minimising wastage is desirable.

This is not an architecture to be employed by novices, the programmers implementing the applications require an in-depth working knowledge of the system. The architecture has been designed to facilitate our research, and we are presenting the concepts to help others develop similar applications.

2 Previous work

A number of previous researchers have identified the need for toolkits and abstractions to help implement VE applications. There are a number of areas that need to be addressed, such as data distribution, rendering, user interaction, tracker abstractions, object extensions, and rapid prototyping. Most previous work only focuses on a subset of areas, leaving the others solely to the implementer. Tinmith-evo5 attempts to provide solutions to all of these areas, as we believe implementing VE applications requires them. This section presents concepts from a number of systems we have referenced during the design process.

2.1 Existing distributed VE systems

A popular area of investigation is designing and implementing distributed virtual environments. This involves operating a software system on multiple processors, and distributing real and simulated entity locations over a network. A focus of these investigations has been on the protocols used to send the information between the processes. Protocols such as SIMNET and DIS (IEEE standard 1278) allow systems to share information with each other, but do not help the programmer implement their software. The type of information sent is restricted to entity objects, containing position, orientation, and related data only. Extensions for features like articulated parts are possible, but not always supported. By making these restrictions however, these systems tend to scale up to large sizes and are more efficient than simpler distribution schemes. Some examples of these message based systems are SIMNET [CALV93] and NPSNET [ZYDA92]. However, the programmer is not provided with support to handle problems like user input, data processing, and rendering. For many applications, programmers want to have the ability to decide this themselves, but for our work we require a more complete approach.

2.2 Existing software toolkits and systems

There are a number of previous systems described (with some being freely available) which attempt to solve some of the areas identified, but do not meet all our requirements.

The ALICE system [PAUS95] is a very high level system allowing novice users to implement simple VE applications. The user can specify object behaviours and interact with them, exploring various possibilities. However, due to the research nature of our work, we wished

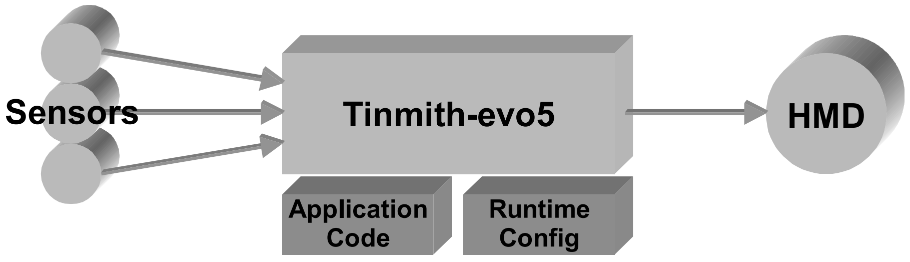

Figure 3 – Overall architecture – Sensors are processed using libraries and application components, then rendered to the user's HMD

to implement new ideas that were previously unthought of, and as a result, making extensions within the existing framework would be difficult.

VB2 [GOBB93] was designed to demonstrate the use of constraints to implement virtual environments. When trackers are updating, the constraint engine manipulates objects in the scene graph, allowing the user to alter the position and orientation of objects. However, constraints are only useful as a solution to propagating data around the system, not covering the remaining problems identified earlier.

One of the most integrated approaches to VEs is COTERIE [MACI96], which was developed to help implement applications for distributed virtual environments. It contains language level support for distributing objects over a network, and integrates this with packages that support an interpreted language, threaded processing, tracker abstractions, and 3D animated rendering. The main focus of COTERIE was implementing distributed systems, and so the shared memory aspect is the main core of the system, with other features built around this. Applications are built up using multiple threads that communicate via replicated shared objects, with each update blocking the thread and passing through a sequencer process that handles synchronisation. This sequencer inhibits the system from scaling up as well as a non-synchronised system. An integrated 3D library and tracker abstraction layer allows the user to implement compiled or interpreted programs to work in AR or VR environments.

3 Tinmith-evo5 concepts

As previously mentioned, we have investigated outdoor AR for a number of years, and have developed a number of systems. This section describes the previous systems we have developed, as well as the concepts used for the new design.

3.1 Initial work

Initially, we produced a simple outdoor navigation system [THOM98] that allows a user to walk through unfamiliar terrain and find waypoints. This was a first attempt at producing an outdoor AR system, with a simple design that polled for serial port data and X events in a single event loop. We knew before the system was finished that we would have to move to a more scalable design if we wanted to build more complex systems.

The Tinmith project began with the goal of creating a design that would allow AR systems to be built from abstracted functional blocks and distributed over multiple machines. The system was built up using ten separate processes, communicating using a data flow model and TCP/IP, that could be distributed if needed, or executed on the same machine. Each process was implemented using functions executed from I/O event callbacks, although the system was implemented with C and not an object oriented language. Tinmith-III [PIEK99c] demonstrated how an outdoor AR wearable computer could locate DIS protocol entities, share location information with other machines, and allow indoor users to use VR displays to visualise the outdoor scene.

As we extended the system, there were problems with having large procedural programs implemented as only ten modules. Since the communication was done at the module level, the modules could not be broken down or combined together. The TCP/IP connections and many executing processes caused the Unix kernel to waste most CPU cycles performing IPC and context switching. Debugging the system was difficult when there were logic errors in any of the modules. Design flaws in the data flow of this version prevented the implementation of complex user interaction without a redesign. As a result, a new design using previously learned lessons was deemed necessary.

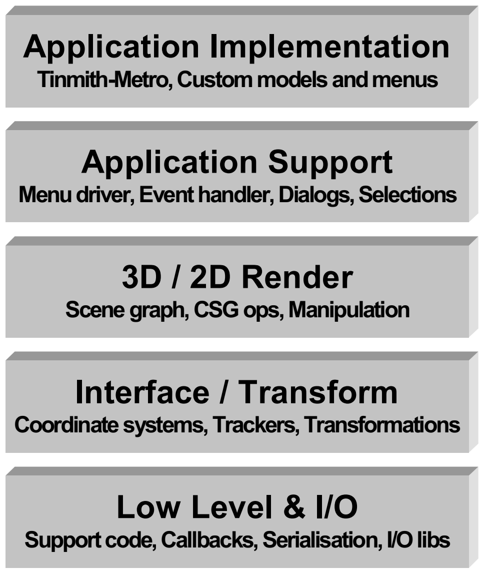
Application Implementation
Tinmith-Metro, Custom models and menus
Application Support
Menu driver, Event handler, Dialogs, Selections
3D / 2D Render
Scene graph, CSG ops, Manipulation
Interface / Transform
Coordinate systems, Trackers, Transformations
Low Level & I/O
Support code, Callbacks, Serialisation, I/O libs

Figure 4 – Library layering

3.2 Tinmith-evo5 concepts

Although the architecture shares the same Tinmith name as previous systems, the version described here, Tinmith-evo5, is based on a completely new object oriented design, implemented in C++. The main goals were speed and efficiency, while allowing us to implement real world applications easily.

The overall design of the system is the data flow model, with objects processing some data and then making it available to others that are interested (similar to an Observer/Observable pattern [GAMM95]). In the previous system, these objects were implemented as a small number of Unix processes. Instead of having processes, objects are now employed, and decompose the problem into hundreds of simple and well-defined tasks. Rather than using costly processes, threads and IPC mechanisms, a single process is used which allow objects to pass information with no overheads. By implementing serialisation objects, it is still possible to distribute the system, but this is an extension and not a required part of the system. Processes can be thought of as containers that allow objects to execute within, allowing them to be moved or reconnected easily.

Given a suitable set of abstraction layers for the operating system and external libraries, the entire system is implemented using data flow interfaces. This allows the system to be consistently implemented, and provides the flexibility to easily make changes. Overall, the system takes in sensor values, processes them inside the Tinmith black box, and writes the results out to a display, as shown in Figure 3.

4 Tinmith-evo5 implementation

The Tinmith-evo5 architecture is based on the basic concepts outlined previously, but to implement this, various interfaces and objects were written. The functionality of the interfaces and objects described in this section is implemented once at a low level so it can be reused consistently. This section discusses the features of this implementation.

4.1 Support libraries

In order to implement the system, numerous support libraries were written to simplify the process. Operations like low level graphics, I/O abstractions, and other simple functions are handled in these libraries. The data flow objects in the system use these libraries to simplify their implementation. Collections of objects are logically placed together into libraries.

```
/* Find source object and attach destination callback to source */
Position *source_position = Position::getStorage ("/dev/trackers/gps/pos");
source_position->setHandler (dest_position->process_position);

/* Make changes to source value */
source_position->setLatitude (138.00);
source_position->setLongitude (34.00);
source_position->setAltitude (0.0);

/* Execute callbacks for interested handlers (dest_position callback is executed)
*/
source_position->callHandlers ();
```

Figure 5 - Code demonstating object store usage and callback linking

Figure 4 shows an overall view of the various components of the system, each building on the previous. At the bottom level are the libraries that abstract away all the non-data flow based hardware and operating system calls. The various levels above contain objects that use instantiated objects at the same or lower levels as their input sources.

4.2 Callbacks

The data flow model is supported by each object implementing callbacks. Objects can register an internal callback method with a desired source object. When the source object changes, the callback method will be executed, allowing the destination object to perform some processing. Figure 5 shows a code fragment setting up a callback, with (1) of Figure 6 showing the relationship graphically.

An object's values may be modified in a number of ways, such as using methods, pointers, or direct memory writing. Tinmith-evo5 does not rely on the compiler preventing access to the object in order to implement change propagation. When the caller is done making changes (in many cases, there are multiple changes made) the object's `callHandlers()` method is executed, indicating the finish of this set of changes. The advantage of this scheme compared to each method call being propagated is multiple changes may be made and then propagated atomically without the problems of synchronisation and locking.

The callbacks are implemented using cpp preprocessor macros, and allow any method or static function to be called when a value changes, with type checking to ensure they are compatible. Each object maintains a list of callbacks to execute when required. Since these callbacks are implemented using function calls and pointers, there are no noticeable overheads imposed.

4.3 Object store

One problem with systems that store large collections of objects is accessing and updating them; the traditional approach being global variables. To overcome this problem, systems such as dVS [GRIM91] and COTERIE [MACI96] implement the concept of a repository where objects can be stored for later retrieval based on a key. The Windows operating system implements a registry, which is a hierarchical database of values stored on disk, used to configure the operation of the system from a central location. Previous Tinmith systems [PIEK99c] used an SQL database to store many configuration parameters, allowing dynamic configuration of various run-time parameters.

Tinmith-evo5 integrates all these concepts into an *object store*. Instantiated objects in the system are created and then stored using a pointer into a global memory structure. Using hierarchical path names similar to that used in a file system, objects can be stored and retrieved by other code easily, without knowing details about the source's creation. The programmer can create objects and a virtual directory structure will be created to store them.

In previous systems, we had a run-time configuration system and wanted to continue with this idea in the new design. The run-time configuration is integrated into the object store for consistency with other system components. Directory structures of text file definitions are stored on the system's secondary storage, with a file extension indicating the type of object. This directory hierarchy is processed when Tinmith applications are initialised, with each text file being read in. Based on the extension, the appropriate object type is instantiated and placed in the object store. Then, the text data is passed to the `fromText()` method of the object, so it can initialise itself with the contents of the file. This method is manually written for any object that participates in the configuration database, and is not a requirement. We manually implement the file formats for human-editable object configuration files as we want to keep them simple and a single generic format would be confusing. One advantage to the configuration system is that the files can be edited using a text editor and when saved, the system re-parses the file and transparently changes the object value in the system. This allows us to experiment with changing system values during execution such as gadget colours and locations, strings, and debugging controls, without the complexities of supporting an interpreted language. Although text files are employed, Tinmith supports the use of alternative storage mechanisms such as SQL databases. Text files were chosen as they are easy to maintain and primitive disk I/O is very simple and fast compared to a heavyweight database process.

Object symbolic links allow the redirection of the flow of data between objects without direct callbacks being attached. The symbolic link object reads a value from one object, copies it, and passes it on to another. The source object can disappear or be reset to another object but the final destination object does not need to take any action. Using this feature, we implement a patch board of tracking devices, allowing us to be flexible with the way we handle the problem of dealing with multiple input devices. An object could be written that monitors GPS and vision trackers, along with a simulator, and redirects a generic tracking object to point to whichever is the most accurate at the time. Although it is possible to implement an object that can simulate this feature, the symbolic link is automatically generated for all objects and is simpler to use.

4.4 Serialisation

Tinmith-evo5 is not based on data distribution in the same way as systems such as COTERIE. However, using the data flow approach, it is possible to add this functionality if required. Figure 6 part (1) shows two objects communicating via callbacks, and using a transmit and receive object inserted in the middle, as in (2) of Figure 6, the data can transmitted

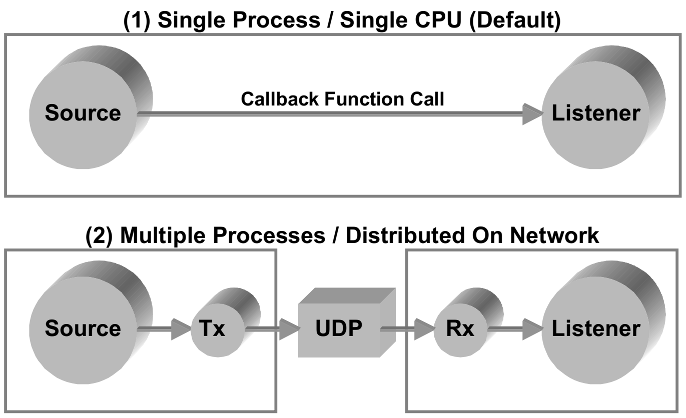
(1) Single Process / Single CPU (Default)
Source
Callback Function Call
Listener
(2) Multiple Processes / Distributed On Network
Source
Tx
UDP
Rx
Listener

Figure 6 – Network distribution is implemented transparently using automatically generated serialisation callbacks

across a network or other IPC mechanism if required. When the `callHandlers()` method is executed, the data is converted to a string with a method call, sent and received using IPC, decoded and stored in a local copy, and then made available to the destination object. Each process maintains its own local copy of the object so that it can be accessed with no blocking or delays when required for use.

Languages such as Java implement serialisation in the language itself, including the ability to send large data structures such as circular rings, trees, and graphical objects. These features would be very useful for our task, but for various reasons we choose not to implement Tinmith-evo5 in Java, such as its resource requirements and performance.

We implemented a serialisation mechanism that would work with C++. The major requirement was to avoid having to manually implement code for all the objects in the system, as this would be tedious and require massive changes if the architecture of the system is modified. The serialisation was implemented with our custom TCC code generator to write these functions (discussed in the next subsection). However, it is possible for the programmer to manually implement serialisation methods if required. We have used this feature in a few cases to serialise objects that are too difficult for the simple TCC program, or where we can implement it more efficiently.

We use a simple serialisation scheme that can be implemented in other languages as needed, allowing other software not written using our architecture to still share information directly with the system. This is in contrast to implementing a program in C++ to be able to read Java's serialised data, which would be a highly complex and time consuming exercise. Many other systems that use complex serialisation schemes are also difficult to interface with.

The serialisation system understands primitive C++ types (int, float, double, string, etc) and also STL containers (vector, list, hash, etc). Objects that implement these types are also serialisable, the TCC program parses the class definition file to make calls to the basic primitive serialisation functions that convert these values to portable strings. The definition files are embedded with cpp macros to make this process easier, also telling TCC which values (such as file descriptors and graphical handles) cannot be serialised. A binary format and Intel little-endian byte ordering is employed to ensure compatibility across multiple CPU architectures if needed.

We use a non-blocking UDP object to perform the transmission of the data over the network, as reliability is not a major requirement in most cases. Tracker updates arrive very quickly, and so if one is lost then the next one can be used as a replacement. In addition, the software processes can be restarted transparently without the need to re-establish a connection. The UDP transport could be easily replaced with a TCP object if required, which guarantees reliability but at a much higher performance penalty. As the serialisation system is not a core part of the system, it can be extended to suit the requirements of an application.

Tinmith-evo5 is client/server based, so one process in the system contains the object, and others can request to have updates sent as changes are made. Resources are conserved by only sending the required data to those objects that are interested. For small systems, this is more efficient than broadcast protocols, although for large systems with thousands of processes each requiring a value, this may not be appropriate. By using proxy processes, cached copies of values may be further distributed to others, allowing the system to scale further.

Our architecture does not contain a single marshalling point or database where all messages must pass through - if the system objects are distributed evenly over multiple processes, then the load for the propagation of data can also be shared as well. A single marshalling server is a performance bottleneck and limits the size a system can scale to, and due to the propagation mechanism used in Tinmith-evo5, is not required.

4.5 Class definition and code generation

C++ supports object inheritance, a preprocessor, and templates, which are very powerful tools, but still have certain limitations for code generation. The TCC code generator was created to write code in cases where we could not do it otherwise. Although it would be possible to implement all the code manually, much of it is tedious, and simple changes to the system or the class itself would require a large number of dependent code changes. Since the system has experienced a number of major changes over its design cycle, minimising these error prone tasks was a major advantage.

TCC processes the definition file for a class, which is used as the authoritative source of information for how to generate the code. Figure 7 shows an extract from the C++ definition file for an IS-300 tracker class, implementing various interfaces. A major advantage is that Tinmith-evo5 does not have a separate interface definition language like used in CORBA or SUN RPC. Instead, the definition is the C++ class definition, eliminating the problem of synchronising code against definition files.

4.6 Run-time and low level support libraries

In order to implement the data flow model, a suitable run-time system, along with code that interfaces to the operating system, is required. Internally, the system is implemented as a single monolithic Unix process, with no threads. The main point of control is the asynchronous I/O manager, which monitors all file descriptors using `select()` for incoming events, and then executes callbacks in objects when they are available for reading. Reads and writes to file descriptors are all done in a non-blocking fashion, and the I/O manager contains buffers to allow the rest of the system to continue operating when the kernel buffers are full. While this approach may be complex to implement, it is not visible to the objects that use it – the advantage is that the code executes in the most efficient manner on the CPU.

We intentionally avoided threads, as they are not useful for our task. In most systems, sensor inputs arrive, are processed, used to affect the state of the system, and then rendered out to the display – all on a single processor. Since a computer can only perform one task at once, and each stage in the pipeline is dependent on the previous, (ie. it can't render the scene graph until the trackers are processed and have modified the scene) a single thread of control is all that is required to process the data sequentially. Using multiple threads wastes CPU resources on context switching and synchronisation. Threads are needed (in the rare occasion)

```
class IS300 : public Storable, public Orientationable, public Matrixable
{
#define STORABLE_CLASS IS300                 // Declare class name
#include "interface/storable-generic.h"     // Include customised template code

public:                                     // Declare callback using macro wrap-
per
   GEN_CALLBACK_H (IS300, process_device, IOdevice, _dev);

   Orientation     *getOrientation (void);  // Implement orientationable interface
   IS300tracker    *getIS300tracker (void); // Access custom IS300 values
   Matrix          *getMatrix (void);       // Implement matrixable interface
   RateTimer       *getRateTimer (void);    // Implement statistics interface

private:                                    // Process dependent variables
   IOdevice *device;                        // Device pointer, not serialised by
TCC

   TCC_OBJ (Orientation,   ori);            // Serialised orientation object
   TCC_OBJ (IS300tracker,  is300);          // Serialise IS300 values
   TCC_OBJ (RateTimer,     rt);             // Serialise statistics information
   TCC_VAR (double,        value);          // Serialise C double value
};
```

when a task must execute independently of the overall event flow of the system, and does not need to update each refresh of the display. In this case, we move the objects into a separate container process and distribute the required data using serialisation. We currently only use this feature for the image recognition component, as the kernel drivers cannot perform non-blocking video I/O, and the process is resource intensive. We wanted this process interleaved with the rest of the system rather than blocking it. The entire system can be threaded or executed sequentially to whatever level is desired and appropriate for the task.

To help with the development of the system, a number of other libraries have been implemented. The fish-malloc library was written to abstract away allocation using malloc/free and new/delete. It assists with debugging and implements a number of special checks at both stages to ensure that memory was not overrun, is released correctly, and does not leak during operation. Other libraries provide similar functions, but not fully integrated in the same way. Fish-malloc can be easily tuned or turned off with preprocessor macros, which have no run-time overhead. This library has proved invaluable in the tracking of problems in the system, as C++ is fundamentally not as strongly typed as other languages, such as Java. An event logging system is included which allows the programmer to generate notices and errors that are written out to the console or a file. When problems occur, these logs (including a GDB stack trace for fatal errors) can be used to locate the source of the error. Apart from using just passive run-time checks, the code was written with safety in mind, so common C/C++ programming problems in areas such as static buffers and dynamic memory allocation have been avoided or protected against in most cases.

5 Sensors and events

Since this system is designed for implementing VE applications, the support for trackers and other input devices is extensive, with a powerful abstraction model and a variety of representation formats.

5.1 Coordinate systems

When navigating on a global scale, latitude and longiture (LLH) spherical coordinates are used to represent locations. GPS trackers also output these coordinates, but in some cases it is desirable to use other systems. For local navigation, flat Earth models like UTM (Universal Transverse Mercator) are useful as it is based on a square grid in metres. In other areas, such as surveying and the DIS protocol, ECEF coordinates (Earth Centred XYZ) are used. The Position class supports the storing and retrieval of any of these types of coordinate systems [ICSM00], with appropriate conversions on demand.

5.2 Abstraction interfaces

Most other VE systems support an abstraction layer for various tracking devices. However, there appears to be a lack of support for representing both absolute and relative devices, as well as representation units. Apart from just implementing a raw Position and Orientation class (along with a 6DOF Tracker combination) we have also implemented PositionOffset and OrientationOffset classes, and the use of units such as metres. These offset classes are used to represent relative shifts that can then be added to an absolute value to obtain a new absolute value. For example, a graphics tablet might be tracked absolutely, but the pen is relative to the tablet. Representing the pen position with a PositionOffset means that we can add this to our current tablet Position/Orientation values, working out the pen position in real world UTM coordinates. We also implement C++ operators to allow the combination of values to produce new results. In the rendering section, we discuss how the scene graph is used to implement similar functionality at an even more flexible level to support articulated parts.

5.3 Resolution problems

Since libraries like OpenGL prefer Cartesian coordinate systems, we try to use position values with UTM because this is the most appropriate match. As a user moves around the world, the origin of the camera moves around as well, and the matrices that perform the rendering are recalculated. If the user moves too far from the UTM origin, these values can become very large, resulting in the values losing resolution when multiplied against large numbers of matrices in the OpenGL pipeline. Millimetre level detailed objects close to the user start to shake and distort, and this causes problems for the user. To overcome this, the rendering system and the Position class work together to produce a new dynamic coordinate system with an origin that is closer to the area the user is operating in (within a few hundred kilometres). This local coordinate system uses smaller values that each object implements, and so the translation is completely transparent to the user - it hidden inside the system and not visible unless required. Using this technique, it is possible for our system to operate over very large coordinate spaces while still handling finely detailed objects, giving it a large dynamic range which is not possible using standard UTM coordinates.

5.4 User interface

In addition to the user walking around and experiencing the VE, the user has to be able to interact fully with the system, entering and manipulating data in the same way as is done on the desktop. Traditional input devices like mice and keyboards are clumsy and inefficient in a mobile outdoor environment, and so we are investigating new and novel user interface technology. The Tinmith-evo5 architecture currently includes support for control menus and object selection and manipulation. The user's input device is a 3D-tracked pair of gloves worn by the user, and this user interface is collectively known as Tinmith-Hand.

Applications that we are building with Tinmith-evo5 have a complex user interface task space. We integrated a 3D pointing device with command entry to improve efficiency. The command options are shown in a menu system placed at the bottom of the display. The user selects an option by making a pinching gesture with the finger that is mapped to the command shown. Some nodes in the menu will move to the next level in the hierarchy, and others will execute a command directly. The menu panel is fixed to the bottom of the display, being visible no matter where the hands or head are located. Our entire system is controlled using the menu and gloves - there is no keyboard attached to the system.

The main application domain of our current work is complex modelling tasks. We want the user to be able to walk outside, and construct models that match real world structures using their hands. Once created, objects need to be manipulated, and therefore selecting them is also required. We support multiple input devices: one or two handed gloves, trackball mouse, and fixed eye cursor.

Selection can be a tedious process if there are large numbers of objects to select, so the user can expand the scope of the selection by moving up the scene graph to select parents of objects, use different camera angles (such as a top down map or orbital view), and take advantage of multiple clipboards. These clipboards allow the user to build up clusters of objects and operate on them as a group, and any number of clipboards are available for use.

Currently, we implement image plane techniques [PIER97] that allow the user to manipulate the objects in the selection buffers. Image plane manipulations treat the objects as flat 2D entities, and map 2D movements of cursors into 3D transformations of objects. Any of the input devices can be used with this, and two-handed manipulation [ZELE97] is particularly useful as the user can rotate or scale the object using one hand as a reference for the other.

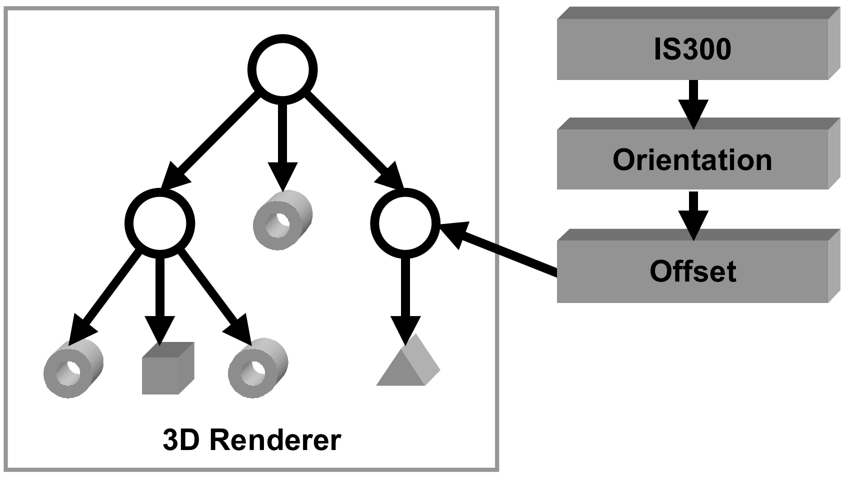

Figure 8 - Tracker to scene graph mapping

6 Rendering system

A major core component of Tinmith-evo5 is the rendering system. It is a full hierarchical modelling system, similar to SGI's Inventor [STRA93] in that it implements a scene graph of objects and transformation nodes.

6.1 Renderer implementation

The scene graph for rendering is integrated in with the rest of the system, so that it can implement serialisation and callbacks. Scene graph nodes are also referenced in the object store, and contain a unique path name so they can be easily referenced elsewhere. Tracker devices can be directly linked to scene graph nodes using a controller interface (see Figure 8), automatically applying movement to the node, including all children. Using this, we can easily implement applications with complex articulated models. In our current system, we include a complete 2 metre human avatar (shown in Figure 10), with 15 separate articulated parts, and we link all our tracker devices into it. Rather than working out complex matrices to find the location of our hands in world space (given coordinates relative to a camera at an arbitrary orientation and position), we use the scene graph to resolve these values for us. Using this, we can graphically verify the tracker motion rather than debugging complex matrix stacks. Scene graph nodes can be used as sources to produce values for other objects in the system.

The renderer uses a native custom model format, and participates in the run-time configuration system, so the world and avatar models are automatically stored into the hierarchy at initialisation using `toText()`. We have written conversion routines to allow us to share information with other modelling systems and formats easily.

OpenGL is used to render the displays, with each object in the scene graph containing a cache of its current generated facet list. Tinmith-evo5 supports complex primitives such as spheres, cones, planes, etc, which are generated once and only regenerated when their defining values are modified. The scene graph is optimised to minimise CPU usage.

6.2 Constructive solid geometry engine

A key renderer feature is the constructive solid geometry (CSG) engine. CSG allows complex shapes to be created using only simple input primitives. Any two Tinmith-evo5 scene graph nodes can be used as an input source to the CSG object, which will produce a new scene graph node that is the boolean combination of the two nodes. The methods are based on similar operations used in ray tracers, such as POV-Ray [POVT00].

Each primitive object is required to be a fully closed surface, with an outside and inside. Infinite planes are also possible, in that the plane cuts the infinite world into an outside and inside space. The CSG engine can test if an object is partially or fully inside a second object, and three fundamental CSG boolean operations possible are demonstrated in Figure 9. Convex objects such as a box can be constructed using six infinite planes and an intersection operation. Combining these operations together may be used to form highly complex concave objects.

The CSG engine operates on facets. Because of this, the CSG engine operates interactively and without loss of detail, which is not possible using ray-tracing or voxel techniques. During a CSG operation, (and whenever a source object is transformed) each source object is broken down into facets and then subdivided, resulting in a new mesh, which is $(\text{input})^2$ in size. Each facet is then tested with the boolean operation and forms part of the output. While computationally expensive, the process still executes at interactive rates under manipulation unless objects with thousands of facets arranged at many angles are used.

7 Applications

An architecture is only useful if there are applications that use its features, and so we have written two demonstration applications (collectively known as Tinmith-Metro [PIEK01b]) which combine together all the Tinmith-evo5 components. The first example allows the user to construct models of outdoor structures such as buildings, and the second allows the user to place down 'street furniture' objects while walking around outdoors.

In both cases, we added special custom accelerators to the menus, which would allow the user to create objects that were customised for the applications. Tinmith-evo5 itself is very generic and allows the user to specify all the information, but also has the ability to use default values to minimise the work the user must do.

7.1 Building construction

Traditionally, AR systems render out models that are initially created on 2D desktop systems. Modelling buildings is tedious and requires the user to switch between desktop (CAD) and outdoor (tape measure and paper) environments to iteratively refine the model until the user finds it acceptable. We wished to be able to enter information about the world into our system in real-time outdoors, using the overlay ability of AR, improving the efficiency of this task by interactively being able to verify the models during modelling.

We believe that CSG is a natural way for humans to understand modelling concepts, as it involves operations such as carving and joining, which is experienced in daily life. This modelling concept (involving new user interaction techniques and displays) was one of the driving factors in the design of Tinmith-evo5.

To model an arbitrary shaped building, we use a number of infinite planes, and then produce a box out of them using the CSG intersection operation. The user must therefore define

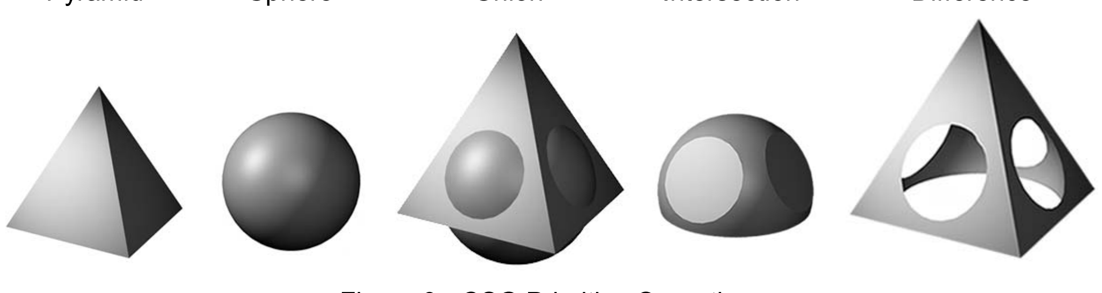

Figure 9 - CSG Primitive Operations

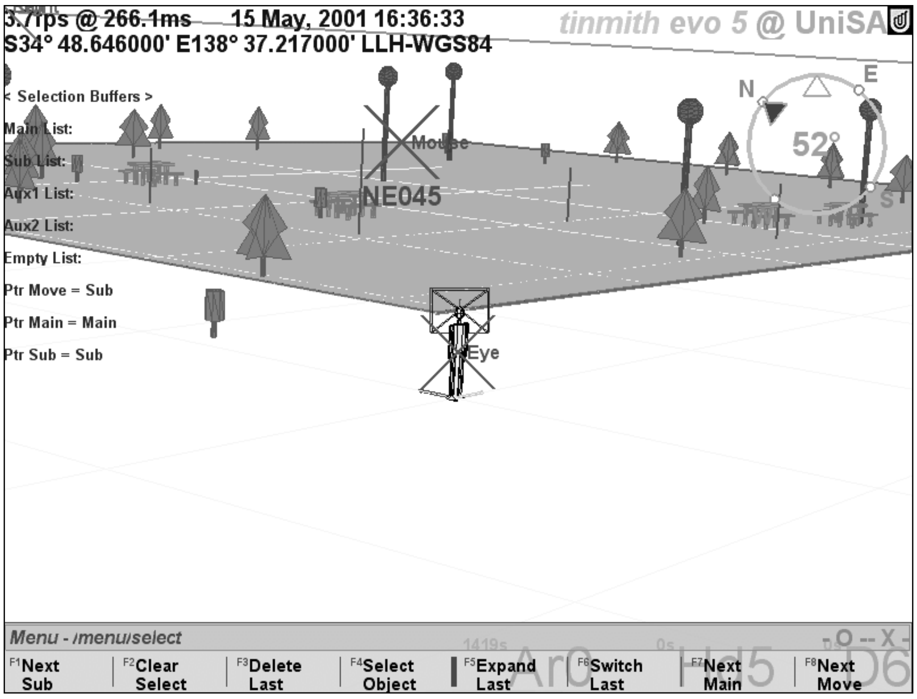

Figure 10 - Tinmith-Metro outdoor furniture placement

these planes and performs the following steps: 1) the user manoeuvres to any location so they are looking down the edge of a building wall, 2) the eye cursor on the display is placed along the wall edge by the user, 3) a right facing wall is created by pinching a menu option, 4) the system creates an infinite right facing plane to the virtual world that intersects the eye cursor and perpendicular to the image plane. By walking around the building and marking each of its defining planes (walls, roof, floor) the user is closing off a portion of the world to form a bounded region. The user can then select the CSG operation to build these objects into a solid object, throwing away the rest of the facet parts that were used to form it. Figure 1 shows an example of a building created using this technique, using only the wall, roof, and floor infinite plane primitives. If desired, other building features such as pitched roofs, windows, and tunnels can be carved using the CSG operations.

7.2 Street furniture

This second example shows how the system is used to implement a 'street furniture' application. The user can use this to place down prefabricated models of existing community infrastructure such as tables, trees, lights, and so forth. We initially created the street furniture objects in NewTek LightWave, and made them available for instantiation into the scene graph by placing them in the object store.

Firstly, we create a grass patch using the previously explained infinite planes technique, which allows the system to represent the ground surface. Next, we walk around the environment (in this case the university courtyard) and pinch the menu options to place the various objects in front of us. We can then use the selection and manipulation techniques, or possibly even a CSG operation, to further refine the model until we are satisfied with its placement. The final example shown in Figure 10 was created in the amount of time it took to walk between the various objects, with the system itself not slowing the user down.

7.3 DIS protocol support

The Tinmith-evo5 architecture still supports the DIS information sharing concepts demonstrated in [PIEK99c], except the new implementation is even more highly integrated and transparent, due to the integrated scene graph support. DIS transmission objects can be attached to the scene graph (or any other suitable object) and generate the required UDP packets when changes are made or after a timeout. As new entities arrive, a node in the scene graph (along with an appropriate model representing it) is instantiated to allow the user to see it, and updated from then on. The asynchronous nature of Tinmith-evo5 allows these packets to be handled transparently.

7.4 Tinmith Hardware

To execute the applications, we use a custom-built backpack computer shown in Figure 11. We use a P2-450 laptop with 64 mb RAM and ATI Rage 3D chipset, as the software has modest requirements on resources. For tracking, we use an InterSense IS-300 for orientation and a Garmin 12XL GPS with differential for position. The display is a Sony Glasstron PLM-700e, mounted to a custom head bracket for mounting the tracker and USB video camera. We use the Linux kernel and libraries, GNU development environment, and XFree86 with Utah-GLX for OpenGL support.

For input devices, we use a set of custom-built pinch gloves with metallic pads that allow the user to control the entire system with hand movements and finger presses. A USB WonderEye video camera and the AR Toolkit software [KATO99] is used to provide glove tracking. A microcontroller is used to poll the gloves and generate serial data for processing in the laptop.

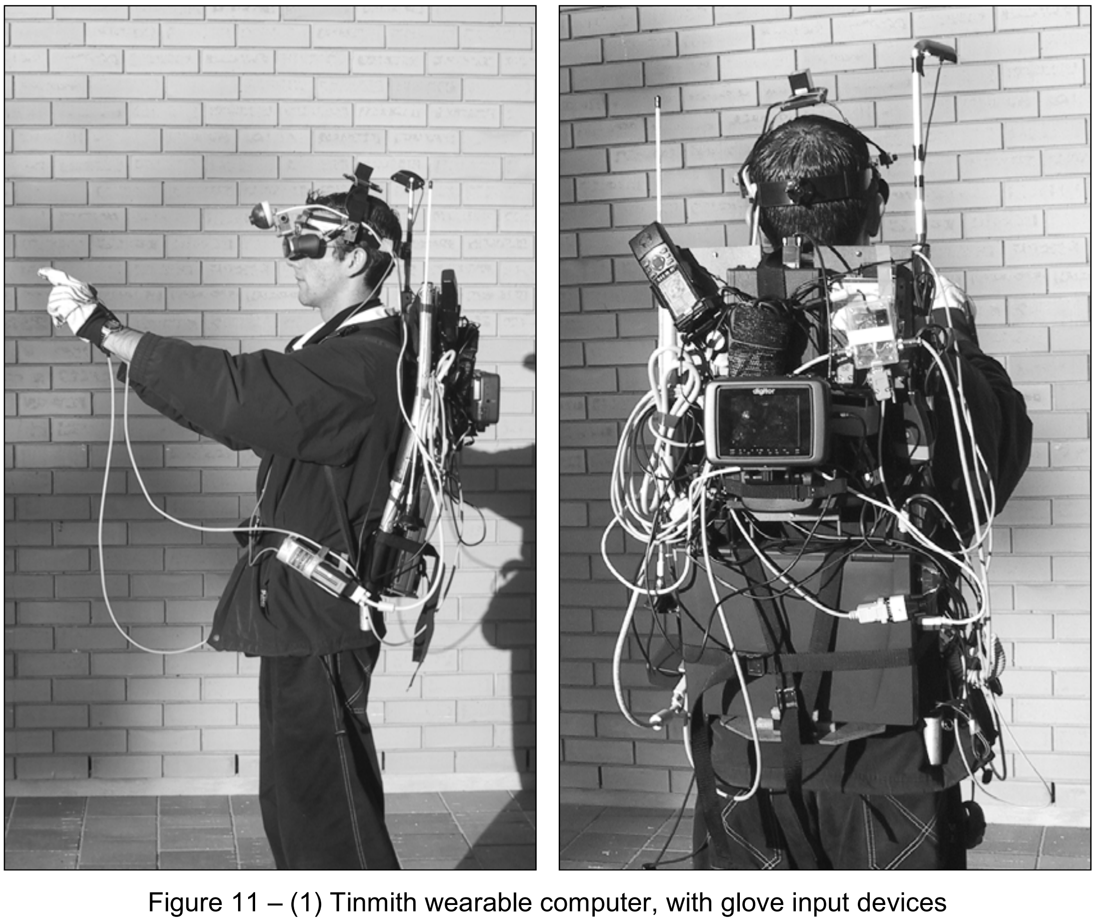
Figure 11 – (1) Tinmith wearable computer, with glove input devices
(2) Rear shot showing various hardware components

8 Conclusion

This paper has introduced the Tinmith-evo5 design, an architecture that provides a uniform approach to implementing virtual environments such as augmented reality. This architecture supports the ability to implement simple object oriented applications that perform a variety of complex tasks, as demonstrated in this paper. Although other systems already exist, we feel that there is not one unique solution to the problem, and that each has its own limitations and advantages. Our system is powerful, robust, highly efficient, and gives the programmer needed flexibility, at the cost of not hiding all implementation details and allowing the breaking of abstraction layers to achieve an objective. It does not attempt to solve all problems, just those that are important for the particular applications we are developing. Using the Tinmith-evo5 architecture, we wish to extend our current modelling applications to build even more powerful systems that can be used in real world environments. By using a flexible architecture, we can extend our design to meet the demands of future, unthought of applications.

9 Acknowledgments

The authors are very grateful for the work of Arron and Spishek Piekarski, who both helped in the design and construction of the glove and HMD. This work was supported in part by grants from the Division of ITEE and the Defence Science Technology Organisation (DSTO).

10 References

[CALV93] Calvin, J., Dickens, A., Gaines, B., Metzger, P., Miller, D., and Owen, D. The SIMNET virtual world architecture. In *IEEE VRAIS '93,* pp 450-455, Sep 1993.

[GAMM95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Ma, Addison Wesley Publishing Company, 1995.

[GOBB93] Gobbetti, E. and Balaguer, J.-F. VB2: An Architecture For Interaction In Synthetic Worlds. In *6th Int'l ACM Symposium on User Interface Software and Technology,* pp 167-178, Atlanta, Ga, Nov 1993.

[GRIM91] Grimsdale, G. dVS - distributed virtual environment system. In *Proc. Computer Graphics 1991 Conference,* London, UK, 1991.

[ICSM00] Intergovernmental Committee On Surveying and Mapping. *Geocentric Datum of Australia - Technical Manual.* URL - http://www.anzlic.org.au/icsm/gdatm/index.html

[KATO99] Kato, H. and Billinghurst, M. Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System. In *2nd IEEE and ACM International Workshop on Augmented Reality,* pp 85-94, San Francisco, Ca, Oct 1999.

[MACI96] MacIntyre, B. and Feiner, S. Language-Level Support for Exploratory Programming of Distributed Virtual Environments. In *9th Int'l Symposium on User Interface Software and Technology,* pp 83-94, Seattle, WA, Nov 1996.

[PAUS95] Pausch, R., *et al. Alice: A rapid prototyping system for 3D graphics.* IEEE Computer Graphics and Applications, Vol. 15, No. 3, pp 8-11, 1995.

[PIEK99c] Piekarski, W., Gunther, B., and Thomas, B. Integrating Virtual and Augmented Realities in an Outdoor Application. In *2nd Int'l Workshop on Augmented Reality,* pp 45-54, San Francisco, Ca, Oct 1999.

[PIEK01b] Piekarski, W. and Thomas, B. Tinmith-Metro: New Outdoor Techniques for Creating City Models with an Augmented Reality Wearable Computer. In *5th Int'l Symposium on Wearable Computers,* Zurich, Switzerland, Oct 2001.

[PIER97] Pierce, J., Forsberg, A., Conway, M., Hong, S., Zeleznik, R., and Mine, M. *Image Plane Interaction Techniques in 3D Immersive Environments. In *1997 Symposium on Interactive 3D Graphics,* pp 39-43, Providence, RI, Apr 1997.

[POVT00] POV-Team. *The Persistence Of Vision Raytracer.* URL - http://www.povray.org

[STRA93] Strauss, P. R. IRIS Inventor, A 3D Graphics Toolkit. In *8th Annual Conference on Object-oriented Programming Systems,* pp 192-200, Washington, DC, Oct 1993.

[THOM98] Thomas, B. H., Demczuk, V., Piekarski, W., Hepworth, D., and Gunther, D. A Wearable Computer System With Augmented Reality to Support Terrestrial Navigation. In *2nd Int'l Symposium on Wearable Computers,* pp 168-171, Pittsburg, Pa, Oct 1998.

[ZELE97] Zeleznik, R. C., Forsberg, A. S., and Strauss, P. S. Two Pointer Input For 3D Interaction. In *1997 Symposium on Interactive 3D Graphics,* pp 115-120, Providence, RI, Apr 1997.

[ZYDA92] Zyda, M. J., Pratt, D. R., Monahan, J. G., and Wilson, K. P. NPSNET: Constructing a 3D virtual world. In *1992 ACM Symposium on Interactive 3D Graphics,* pp 147-156, Cambridge, MA, Mar 1992.